A Three-Tier Approach for Composition of Real-Time Embedded Software Stacks

Frédéric Loiret[1], Lionel Seinturier[1], Laurence Duchien[1] and David Servat[2]

[1] INRIA-Lille, Nord Europe, Project ADAM
Univ. Lille 1 - LIFL CNRS UMR 8022, France
{frederic.loiret | lionel.seinturier | laurence.duchien}@inria.fr
[2] CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems,
Point Courrier 94, Gif-sur-Yvette, 91191, France
david.servat@cea.fr

Abstract. Many component models and frameworks have been proposed to abstract and capture concerns from Real-Time and Embedded application domains, based on high-level component-based approaches. However, these approaches tend to propose their own fixed-set abstractions and ad-hoc runtime platforms, whereas the current trend emphasizes more flexible solutions, as embedded systems must constantly integrate new functionalities, while preserving performance. In this paper, we present a two-fold contribution addressing this statement. First, we propose to express these concerns in a decoupled way from the commonly accepted structural abstractions inherent to CBSE, and provide a framework to implement them in open and extensible runtime containers. Second, we propose a three-tier approach to composition where application, containers and the underlying operating system are designed using components. Supporting a homogeneous design space allows applying optimization techniques at these three abstraction layers showing that our approach does not impact on performance. In this paper, we focus our evaluation on concerns specific to the field of real-time audio and music applications.

1 Introduction

Component-Based Software Engineering is nowadays applied for a wide range of application domains, from IT systems using mainstream component technologies such as EJB or CCM to real-time and embedded (RTE) systems. Beyond the well-established advantages in terms of packaging and composability of independently-implemented software modules, CBSE promotes *flexible design approaches*, relying on separation of concerns embodied by the software's architecture. Moreover, the capability to specialize the architecture with relevant abstractions and non-functional concerns of an application domain, conduct the definition of Domain-Specific Component Frameworks (DSCF). Thereby, DSCF offers a domain-specific component model and a tool-support allowing these non-functional concerns to be deployed in the runtime platform composed of a set of custom made containers. It is especially the case for RTE Component Frameworks relieving the developer from dealing with redundant and error-prone tasks such as creating threads, managing synchronization or activation of the

components with temporal constraints, or performing inter-task communications (ITC).

Problem statement. Many RTE Component Frameworks have been proposed to address these non-functional concerns [6, 9, 24]. However, these propositions tend to provide their own abstractions, fixed set of execution and communication models, and their own ad-hoc runtime platforms. We believe that proposing more flexible solutions is a key issue to consider in order to improve reuse and integration of legacy solutions. Indeed, *i)* components can be independently deployed in heterogeneous execution contexts depending on embedded or temporal requirements. *ii)* Runtime platforms must be adapted according to new non-functional concerns as embedded systems must constantly integrate new functionalities. However, RTE constraints are tightly dependent on the runtime platform and on the underlying operating system since these layers must not induce a significant overhead concerning critical metrics of the domain, such as memory footprint, real-time responsiveness and execution time. The respect of these constraints is thus a prerequisite for introducing flexibility in embedded software stacks.

Contributions. To address these challenges, the contribution of this paper relies on two parts which are integrated into a full-fledged framework. First, we propose the use of a *generic* component-based framework *extensible* towards various domain-specific concerns [13]. These concerns are specified by the developer in a flexible way via the use of annotations specializing the architectural artefacts (components, interfaces, bindings), according to the execution contexts required by the application. Our framework relies on an approach for generation and composition of component-based containers implementing the runtime platform which fits the application's requirements. Second, we exploit a component-based implementation of a real-time operating system presented in [14] providing the low-level services required by the containers. As a result, we present in this paper a three-tier approach where an embedded software stack made of component-based application, containers and operating system is composed using a homogeneous component model. A homogeneous support of the component paradigm is the key point of our contribution to support flexibility into the software stack at these three abstraction levels. Moreover, *i)* by composition, only the services strictly required by the whole system are deployed in the final executable to fulfill the embedded constraints. *ii)* Our approach relies on optimization techniques which are applied uniformly at these three abstraction layers reducing at a negligible level its impact on the performances. In this paper, we apply this approach to the design of real-time audio applications.

Outline. The paper is structured as follows: The two building blocks on which our contribution relies are presented in Section **2**, and a general overview of our approach is outlined in Section **3**. Section **4** presents our contribution to designing component-based applications dedicated to real-time audio concerns. The composition process involved to implement these concerns are detailed in Section **5** and evaluated in Section **6**. Finally, we discuss related work in Section **7** before concluding and drawing future directions of our research in Section **8**.

2 Background

In this section, we present two building blocks on which the contributions of this paper rely.

2.1 Hulotte Component-Based Framework

HULOTTE [13] defines a component metamodel introduced in Figure 1. This metamodel is based on general CBSE principles [4], and is inspired by the reflective FRACTAL component model [3]. In particular, HULOTTE identifies as core architectural artefacts the concepts of `Component` (either `Primitive` or `Composite`), `Attribute`, `Interface`, and `Binding`. The behavior of a primitive component is implemented by the underlying programming language supported by our framework (the C language in the context of this paper) and is reified by the `Content` artefact. An architecture is then specified as a set of interconnected components (at an arbitrary level of encapsulation by using the composite design pattern) via oriented relationships between `Interfaces`.

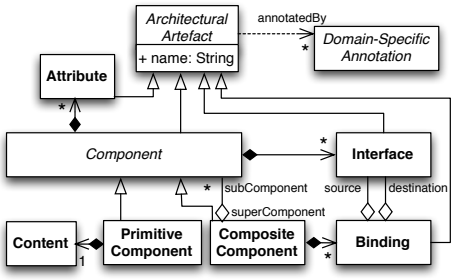


Fig. 1. The HULOTTE Metamodel.

We distinguish two roles involved in the HULOTTE development process: the *application developer* and the *platform developer*. The ***application developer*** is responsible for the development of *applicative components* and the specification of their domain-specific requirements. She/he uses the HULOTTE metamodel concepts, depicted in Figure 1, to design the component-based application, which is afterwards annotated by Domain-Specific Annotations. These annotations mark the HULOTTE Architectural Artefacts like Java 5 annotations mark the *Abstract Syntax Tree* (AST) of a Java program. HULOTTE annotations isolate and specify the concerns relevant to a targeted application domain, so-called *domain-specific concerns*. Within our approach, it should be noticed that components are used as pure business units, and a component-based architecture then implements the whole business logic of the application. Therefore, annotations are used to specify the domain-specific semantics over the architecture. For instance, in order to address the multitask applications domain, an annotation can be used on a component to qualify under which *execution model* its business interfaces should be invoked, *e.g.* periodically or sporadically. As a second example, on a composite component, an annotation can qualify the boundary of a memory scope in which its subcomponents will be allocated. Finally, an annotation can also be used on a binding to specialize the *communication models and protocols* (*e.g.*, asynchronous, shared memory, CORBA, SOAP) between the components it binds.

The role of the ***platform developer*** is to design the runtime platform implementing the domain-specific requirements specified by the application developer. HULOTTE relies on a reflective architecture where each applicative component is hosted by a container, which is itself implemented as a component-based architecture. Throughout this paper, we will refer to "*platform components*" encapsulated within the containers. Therefore, a container is also implemented using the architectural concepts presented in Figure 1.

2.2 Real-Time Operating System Componentization

In [14], we have conducted a component-based reengineering of μC/OS-II [2], a preemptive, real-time multitasking kernel for microprocessors and microcontrollers. It is implemented in ANSI C and certified by the FAA[3] software intended to be deployed in avionics equipment. It has been massively used in many embedded and safety critical systems products worldwide. μC/OS provides the basic services of a Real-Time Operating System (RTOS): Task and Time management, Inter-Process Communications (mailbox, queues, semaphore, mutex), and Interrupts management. The execution time for most of these services is both constant and deterministic. μC/OS is implemented as a monolithic kernel, *i.e.* it is built from a number of functions that share common *global variables* and *data types* (such as *task control block*, *event control block*, etc). The reengineering presented in [14] consists of a library of ready-to-use RTOS configurations, implemented by a set of composite components, providing their services to the application. We showed that overheads involved by our component-based design in term of performance were negligible compared to the original monolithic implementation of the kernel.

3 General Overview of the Approach & Case Study

This section presents a general overview of our three-tier approach for composition of a real-time embedded software stack, presents the application domain and the case study on which it is applied in this paper.

3.1 General Overview

The software stack is sketched in Figure 2. At the higher abstraction level, the application developer specifies the architecture of the application as a set of components annotated by domain-specific annotations. In the context of this paper, annotations are used for designing real-time audio applications and will be presented in Section 4. As a first stage of composition (Fig. 2(1)), these components are composed within component-based containers implementing the semantics of the domain-specific annotations in a transparent manner for the application developer. The implementation of containers by the platform developer and the composition rules between *applicative* and *platform components*

[3] Federal Aviation Administration.

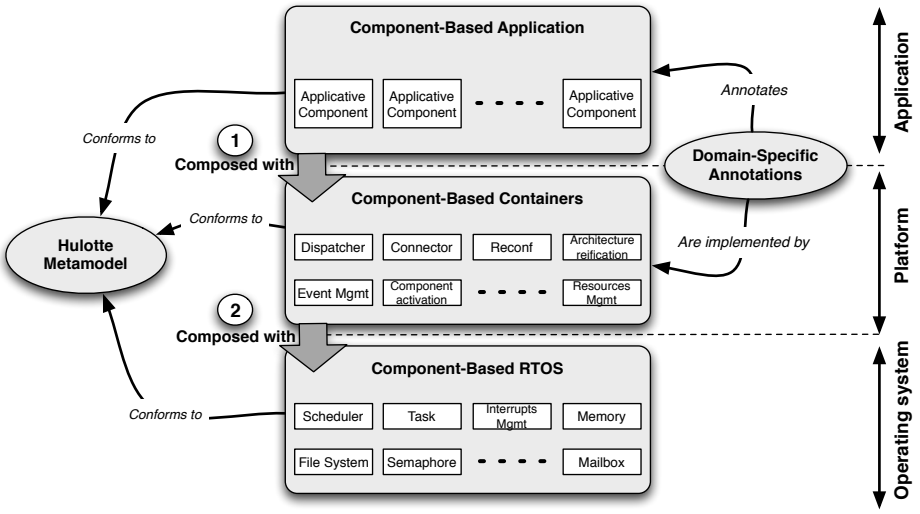


Fig. 2. A Three-Tier Approach for Composition of a RTE Software Stack.

are integral parts of the HULOTTE framework and will be detailed in Section 5. Containers implement non-functional services required by the applicative components, control them, mediate their interactions with other applicative components and with the operating system. Therefore, as a second stage of composition (Fig. 2(2)), the operating system services required by the containers are bound to the component-based operating system presented in Section 5. A detailed description of the roles involved within the design process presented in this paper is sketched out in Figure 3, where steps (1) and (2) refer to the same two composition steps from Figure 2.

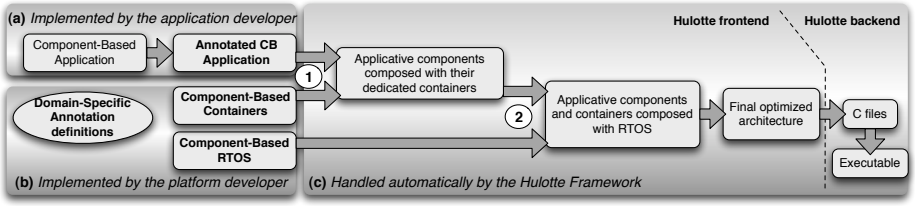


Fig. 3. Description of the HULOTTE's Design Process.

3.2 Specificities of RTE Audio Applications

In this paper, we present our approach in order to provide to the application developer a design space for component-based audio and music applications. These applications are inherently designed as multitask and concurrent softwares since they generally implement audio flows processing algorithms controlled by the

end-user via HMIs (Human-Machine Interfaces). Moreover, they must be executed in a real-time context since audio data must be processed in time to avoid buffer underflows. Therefore, the developer must properly manage the resources of the system (tasks, audio buffers, mutex, etc), implement the audio data copies throughout, for example, a pipe of audio filters – potentially at different sample rates – or the way tasks are synchronized and shared data are protected within critical sections. These aspects are typical domain-specific concerns since from one application to another, their implementations are redundant, time-consuming and error-prone. Therefore, the aim of Hulotte is to provide a design space where these concerns are handled automatically by our framework.

3.3 Case Study: a Vinyl Multimedia Controller

To illustrate the main features of our approach, we introduce our case study (DeckX) on which we will rely throughout the paper: An application for DJ's that uses a special vinyl record as a mean of controlling various multimedia sources (such as MP3 audio or video files) from a classical turntable. It operates as follows: the vinyl is pressed with a dedicated sinusoidal stereo signal which encodes a "time-code". A software analysis of the signal gives three pieces of information: the absolute position of the turntable arm in the vinyl, its velocity and its rotation direction. Concretely, it is thus possible to "scratch" in real-time MP3 files stored in a computer from the turntable. Moreover, we consider the ability for the DJ to control audio parameters (output volume and filtering parameters) from the keyboard. Our case study thus represents a real-life application, and is composed of various functional parts which have to meet various concurrent constraints. Moreover, the application is intended to be deployed in a microcontroller and must therefore fulfill constraints encompassed by ressource limited devices.

4 Application Level: Designing RTE Audio Applications

This section presents the domain-specific annotations provided to the application developer for designing real-time and audio applications. We outline then how DeckX is implemented with Hulotte.

4.1 Domain-Specific Annotations

The audio-domain-specific concerns presented in Section 3.2 are modeled by Hulotte annotations. These annotations are used by the application developer to specialize its application specified as a set of interconnected applicative components. The list of annotations provided to the developer is given in Table 1 and their basic semantics are detailed below.

`@ClientServerItf` and `@AudioItf` annotations specify the signatures and properties of the interfaces exported by the components. A *Client-server interface* signature defines a set of method signatures (like Java interfaces) with a list

Table 1. HULOTTE Annotations Dedicated to Real-Time Audio Applications.

Annotation	Applied to	Parameters
@ClientServerItf	`interface`	**signature**: string **role**:{client | server} **cardinality**:{single | multicast}
@AudioItf	`interface`	**signature**: string **role**:{producer | consumer} **cardinality**:{single | multicast}
@Buffered	`interface`	**bufferSize**: integer
@MonoActive	`interface`|`component`	**priority**: integer
@MultiActive	`interface`|`component`	**priority**: integer **poolSize**: integer
@Asynchronous	`binding`	
@Protected	`component`	**initialValue**: integer
@CpuItHandler	`interface`	**irqNumber**: integer
@OSItf	`interface`	

of parameters and a return type. The parameters of the annotation specify if the interface is `client` or `server`, `single` or `multicast`. The `multicast` property specifies a one-to-N connection scheme between one client and N server interfaces. Client-server interfaces model services *required* or *provided* by the components. The *audio interfaces* (*i.e.*, the interfaces annotated with `@AudioItf`) model streams of audio data `produced` or `consumed` by the components. Their signatures define a set of parameters qualifying audio streams: the number of audio frames transmitted, their data types (an audio frame is stored as an integer, a float or a double), the number of audio channels (e.g. mono or stereo), the sample rate of the audio flow, and the way multiple channel frames are intertwined. The need to qualify these audio interface signatures is justified for composing independently implemented components. Indeed, interconnected components may have been implemented according to audio streams encoded differently. An example of a client-server and of an audio interface signatures using the HULOTTE's IDL (*Interface Description Language*) is given in Figure 4. The bindings between applicative components are specified between a client and a server interface or between a producer and a consumer audio interface.

```
public cltsrvinterface
deckX.api.Track {
  struct_track_t *getTrack(void);
  int track_import(char *path);
  int track_handle(void);
}
```

```
public audiointerface
audio.api.AnalyserAudioType {
  audio_frame_size: 64; sample_type: float;
  channels: 2; sample_rate: 44100;
  intertwined_samples: true;
}
```

Fig. 4. IDL's Examples of a *Client-Server* and *Audio* Interface Signatures.

`@Buffered` annotations can be applied to interfaces. Such an interface specifies that service invocations or audio streams passing through it are buffered. The size of the buffer is specified by the `bufferSize` parameter of the annotation. *Buffered interfaces* are for instance used when components produce and consume audio streams at different frequencies.

The `@MonoActive` annotation specifies that an applicative component is attached to its own thread of execution defined with a `priority`. The execution

model attached to such a component performs the incoming *activation requests* sequentially – *i.e.*, in a run-to-completion mode – with a FIFO ordering policy. In our case, activation requests can be operation invocations from a server interface or audio streams consumed from an audio interface. The `@MultiActive` annotation has the same semantics but for a pool of threads performing activation requests in parallel.

The `@Asynchronous` annotation can be applied to bindings. In this case, the thread of control originating from the source interface of the binding returns without being blocked until the completion of the execution at destination interface side.

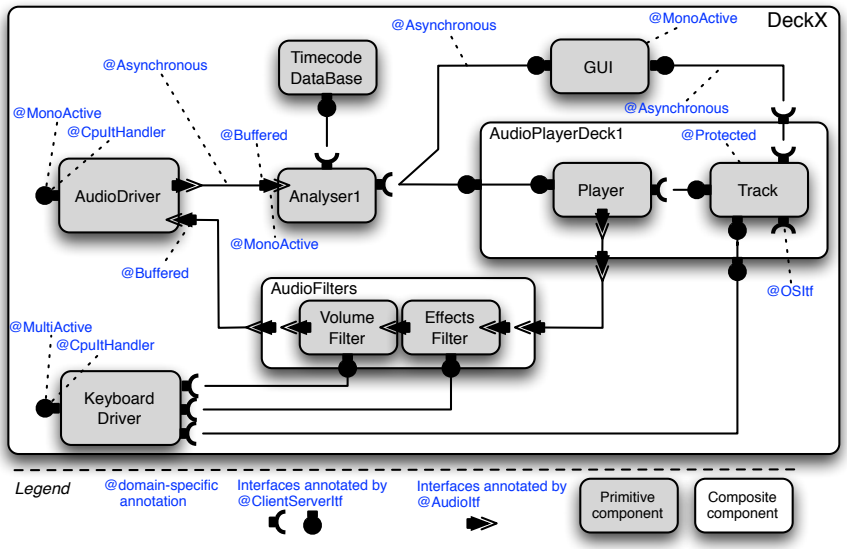
If the implementation code of an applicative component is stateful and not reentrant (*i.e.*, it can not be safely executed concurrently), the developer uses the `@Protected` annotation. In this case, the HULOTTE execution framework guarantees mutual exclusion: a single execution flow is allowed to execute the code encapsulated by such a protected component (just as the `synchronized` keyword in Java).

`@CpuItHandler` and `@OSItf` are used to specify a link between the application layer and the operating system layer. Indeed, two cases are identified: First, an applicative component's execution may be triggered by the reception of a hardware interrupt (`@CpuItHandler`) managed by the operating system. Second, an applicative component may directly require services implemented by the operating system or device drivers not directly handled by the application (`@OSItf`). In these two cases, the annotated interfaces will be automatically bound afterwards by the HULOTTE process as described in Section 5.

4.2 Annotated Architecture of DeckX

The HULOTTE architecture of our case study is given in Figure 5. In the following, we outline its functional and *non-functional* behavior, the latter being specified by annotations: The `AudioDriver` component will be *attached at runtime by a thread of execution activated* by an interrupt managed by the operating system. This driver reads the audio buffers from the audio device (corresponding to the timecoded signal read from the turntable) and *produces asynchronously* audio frames on its outgoing audio interface. The `Analyser1` component is also *attached to its own thread of execution and is activated each time* new audio frames are *buffered* in its incoming audio interface. At each analysis step, timecode information decoded from incoming audio frames are *multicasted* to the `GUI` and the `AudioPlayerDeck1` components. According to this information, the `Player` component processes audio frames read from an MP3 file managed by the `Track` component, sends them to the `AudioFilters` and finally to the incoming audio interface of the `AudioDriver`. The `KeyboardDriver` component *is activated each time* a key is pressed by the end-user and dispatch the process of the corresponding event to change a parameter of the `AudioFilters` or to load a new MP3 track via the `Track` component (the latter thus requires an `OSItf` to access to the file system managed by the operating system layer). The latter is *protected* since concurrent execution flows initiated from the `Analyser1`

or the `KeyboardDriver` components access to its provided services. *The priorities* of the `@MonoActive` and `@MultiActive` annotations attached to applicative components are specified as follows: `Prio(AudioDriver) > Prio(Analyser1) > Prio(KeyboardDriver) > Prio(GUI)`, since audio samples must be handled in priority by the `AudioDriver` and the analysis process, compared to the interactions with the end-user with the keyboard and the GUI. Annotations are implemented by the HULOTTE platform as it is presented in the next section.




```
1 component DeckX {
2   component Analyser1 {
3     @Buffered(bufferSize="512")
4     @MonoActive(priority="20")
5     @AudioItf(sign="audio.api.AnalyserAudioType", role="consumer",
6       cardinality="single")
7     destination interface inputAudio

9     @ClientServerItf(sign="deckX.api.TimeCodeType, role="client",
10      cardinality="multicast")
11    source interface outputTimecode

13    @ClientServerItf(sign="deckX.api.tcDBType", role="client",
14      cardinality="multicast")
15    source interface tcDataBase

17    content AnalyserImpl.c
18  }
19  @Asynchronous
20  binds AudioDriver.outputAudio to Analyser1.inputAudio
21  binds Analyser1.tcDataBase to TimecodeDataBase.tcDataBase
22  // ... Other bindings and components
23 }
```

Fig. 5. Graphical and Textual Representations (excerpt) of a DECKX's Annotated Architecture.

5 Hulotte Platform & Operating System Compositions

As sketched out in Figure 3(b), the *platform developer* is responsible for implementing the runtime platform supporting the domain-specific requirements specified as HULOTTE annotations. The HULOTTE platform is engineered with component-based containers, which brings three significant advantages: *i)* the platform developer benefits from a component-based design to implement the semantics of arbitrary complex domain-specific annotations, in a *decoupled way* from the application logic, *ii)* our approach relies on a reflective architecture, in a symmetric and isomorphic way, and *iii)* the low-level services required by the platform and provided by the operating system are explicitly specified. The concept of container on which the platform is built is generalized, defining composition rules and architectural invariants as architectural patterns to specify the link between applicative components and platform components as it is presented in the following section. The component-based implementation of the operating system is detailed in Section 5.2, as the composition between the container and the OS layers.

5.1 Component-Based Containers Design and Composition

The composition of the containers corresponds to the first *composition step* of the process depicted in Figure 3(c), and relies on a generative and aspect weaving technique. Each HULOTTE annotation is implemented by the platform developer as architectural fragments made of *interceptor* and *platform components*. The platform developer implements a HULOTTE plugin which provides the way these fragments will be woven into composite containers, according to the annotations specified at the application level. The output of this composition step is an architecture description where all applicative components are encapsulated within their dedicated containers. Therefore, for a given applicative component, its container:

- implements non-functional services it requires via annotations,
- mediates the domain-specific interactions with its environment,
- manages the resource instances it requires, such as tasks, semaphores, buffers or message queues, reified as components [14],
- allows *inversion of control* based on interception of execution flows transiting via applicative component's interfaces.

The platform components are generic components provided by the HULOTTE component library, or generated components. The architectural specification and implementations of the latter are respectively generated programmatically and by source code templates according *i)* to interface signatures of the applicative components they will control, and *ii)* the annotations parameters set by the application developer. Interceptors are also generated since their specifications rely on applicative components' interfaces always known only at weaving time. According to the annotations presented in Table 1, the contents of the containers depend on the following information which are specified by the application developer:

- `Buffered` interfaces are intercepted, and data transiting through them (*e.g.*, parameters of method or audio flows) are stored in a buffer implemented as a platform component.
- The semantics of `MonoActive` and `MultiActive` components are implemented by OS task instances handled by generated platform components. The latter control their *thread(s) of execution* as it was mentioned in Section 4.1.
- The logics behind `Asynchronous` bindings and `multicast` interfaces are implemented by interceptors.
- The semantics of `Protected` component is implemented by interceptors, all together bound to a semaphore. The counter of the semaphore is then incremented when an execution flow from the environment of the protected component execute a service it provides and is decremented when it returns.
- In the case of applicative components bound via audio interfaces with different signatures, a dedicated interceptor is generated implementing the conversion algorithm between the source and the destination of the audio flow.

As an example, Figure 6 shows the container of the `Analyser1` applicative component, according to the annotations specified by the application developer presented in Figure 5. Note that HULOTTE annotations are also defined to characterize the specificities of the container level concepts.

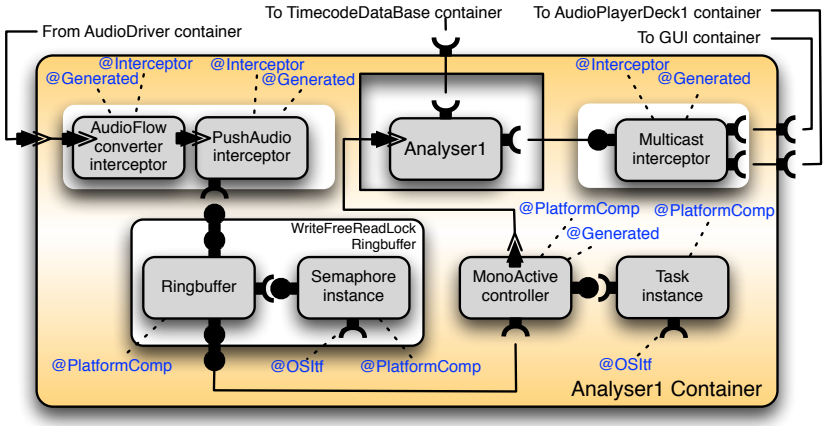


Fig. 6. The Generated Container of the `Analyser1` Applicative Component.

Within this container, audio flows coming from the `AudioDriver` component are converted according to the flow parameters expected by `Analyser1` (specified by the signature of its incoming audio interface) and are buffered by the `PushAudio` interceptor. The multicast client interface named `outputTimecode` (see Fig. 5 lines 9-11) and bound to `GUI` and `AudioPlayerDeck1` components is handled by the `Multicast` interceptor. The semantics of the `Asynchronous`, `MonoActive` and `Buffered` annotations attached to the incoming binding and

interface of the `Analyser1` component (see Fig. 5 lines 3-7 and 19-20) is implemented by the set of platform components encapsulated within the container.

5.2 RTOS Design & Composition

As it has been presented, HULOTTE containers implement non-functional concerns required by the applicative components. These concerns may require operating system services, such as task scheduling, time or Inter-Process Communications management. In this sense, containers act also as an intermediate abstraction layer between the operating system and the application layers.

As it is presented in details in [14], our component-based RTOS consists of a set of primitive components encapsulated within a composite. The latter exports the public services invokable from container and applicative components. Within the HULOTTE process depicted in Figure 3, the second *composition step* consists in a two-direction composition between these layers, via client-server bindings:

- From container (or applicative) components to the RTOS. In this case, client interfaces annotated by `@OSItf` annotation are bound to the corresponding interface of the RTOS.
- From the RTOS to the container components when the latter encapsulates applicative components annotated by `@CpuItHandler`. In this case, applicative components export handlers to serve hardware's interrupts.

This composition step is automatically handled by HULOTTE and is based on the signatures exported by the interfaces. The content of the RTOS is therefore automatically built according to the services strictly required by the applicative and container components.

6 Evaluation

In this section, we provide a detailed evaluation of our approach, from a qualitative and a quantitative point of view.

6.1 Qualitative evaluation

Application's Design Space. HULOTTE provides a component-based design space which enforces a strong separation of concerns. Indeed, the developer is exclusively focused on the implementation of its applicative architecture, afterwards annotated with domain-specific/non-functional concerns. In consequence, this separation occurs also at code level, the latter becoming more readable and maintainable – reflecting the functional needs of the application without any constraints imposed by the low-level real-time audio properties, as it has been experimented with DECKX. Moreover, the decoupling between the architecture and the annotations improves reuse since components can be independently deployed in various execution contexts without any applicative code refactoring.

However, since we propose a generic mechanism where each architectural artifact can be annotated by arbitrary annotations, their use imposes several constraints for the application developer. Indeed, annotations may be applied incorrectly to an artifact, a set of annotations can be self-contradictory and cannot be composed together, or annotations can depend on each other. To tackle with issue, we chose a defensive approach based on constraints (using the OCL constraint language [19]) which must be explicitly specified by the platform developer, and which are checked automatically from an annotated application. These checks operate just before the container generation step and ensure the consistency of the application's specifications. These points are reported in details in [18].

Platform's Design Space. The HULOTTE platform consists of a set of containers implementing real-time-audio annotations. The containers composition process relies on the incoming and outgoing interaction points externalized by the applicative components, through explicitly defined and stable interfaces, and therefore independently of their implementations. Moreover, a strong separation of concerns is applied between applicative and platform components which are linked together by composition without any dependency on the internal elements of the applicative code. These characteristics of our approach allow us to improve sorely the extensibility of the platform, towards the support of new non-functional concerns. For instance, our component-based platform model has been validated in studies spanning various domains, from distributed reflective and reconfigurable applications [13], to *Real-Time Java Specifications* (RTSJ) [21].

Taking Benefits of a Full Component-Based Approach. We can witness several benefits in using a homogeneous component model for constructing RTE software stack, made of applicative, platform, and RTOS components. First, we rely on homogeneous composition techniques based on the component requirements exposed at architectural level to obtain the final stack. As it has been presented in Section 5.2, this feature allows us to built automatically an operating system fitting exactly the services required by the whole system. Second, since the HULOTTE process outputs a flattened architecture of the complete software stack, we can apply uniformly tools based on the abstractions of our component model.

Two features provided by HULOTTE are presented and evaluated thereafter: the capability to support introspection and reconfiguration of the system at runtime and the support of optimization techniques of the final executable. The latter is a mandatory requirement raised from the embedded domain, since relying on high-level abstractions at design time must not impact drastically on performance at runtime.

6.2 Quantitative Evaluation

As illustrated in Figure 3, the HULOTTE framework consists of a frontend implementing the composition steps described in Section 5 and a backend. In this

paper, the backend relies on the THINK component framework [5]. From the flattened architecture outputed from the frontend, the backend generates a set of C source files compiled afterwards by a classical C compiler. In the following section, we measure how our approach impacts the resulting executable in terms of memory footprint and execution time, based on the DECKX case study.

The comparisons are established between a *reference* implementation against a *component-based design*, the latter being based on HULOTTE. The reference implementation corresponds to a version of DECKX where applicative functionalities and these implemented by the HULOTTE containers are implemented manually in a full code-centric approach, and linked to a monolithic implementation of the operating system. In both cases, the same set of functionalities are embedded in the final binary.

For the component-based design, we consider three scenarios: *i)* `Flexible`, where all components outputed from the HULOTTE process are generated as introspectable and reconfigurable entities at runtime. This feature is supported by THINK, which generates meta-data and provides these capabilities at runtime. *ii)* `Not flexible`, which generates a static binary of the whole architecture, not introspectable and reconfigurable anymore. This scenario relies on THINK optimizations described in [12, 14] to control performance overheads induced by the backend framework. Finally, *iii)* the `Flattened` scenario, consisting of a binary generated as static and without hierarchical encapsulation. These scenarios are taken *automatically* into account within the last stage of the HULOTTE frontend process depicted on Figure 3, according to the developer preferences.

Memory Footprint. Figure 7 presents the memory footprints of the reference implementation and the component-based designed for the three aforementioned scenarios[4]. We measure the overhead in code (*i.e.*, `.text` section) and data, including initialized (*i.e.*, `.data` section) and uninitialized (*i.e.*, `.bss` section) data. We make this distinction as code is usually placed in ROM, whereas data are generally placed in RAM. The overheads are not negligible for the `Flexible` scenario (Fig. 7(b)), reflecting the cost to provide a full reconfiguration support at runtime. However, since this feature may not be required for the whole embedded stack, HULOTTE relies on the mechanisms provided by THINK to specify whether a single component or a subset of the architecture should be generated as reconfigurable [14]. When considering a complete static system (Fig. 7(c)), just a code overhead is observed, becoming negligible in the last scenario (Fig. 7(d)). The latter overhead comes for the resource instances reified as components within our approach.

	`Reference`	**Component-Based Design**		
	(a)	**(b)** `Flexible`	**(c)** `Not flexible`	**(d)** `Flattened`
Code	26994	**+20.0 %**	**+5.9 %**	**2.1 %**
Data	17320	**+12.0 %**	**+0 %**	**+0 %**

Fig. 7. Memory Footprint Sizes of DECKX (in Bytes) and Overheads.

[4] These experiments have been conducted using GCC with the `-Os` option that optimizes the binary image size.

The Figure 8 presents the memory footprints for the `Flattened` scenario compared between the RTOS, the platform and the application layers[5]. These results show in particular the important part of DeckX related to audio and real-time concerns, which are automatically handled by our approach (Fig. 8(c)) in an oblivious manner for the application developer.

	Total	Abstraction Layers		
	(a)	(b) RTOS	(c) Platform	(d) Application
Code	27558	41.1 %	**19.5** %	28.6 %
Data	17320	86.2 %	**9.8** %	2.7 %
Comps		11	**27**	14

Fig. 8. Memory Footprint Sizes (in Bytes) and number of components of DeckX for each Abstraction Layer.

Execution Time. Finally, Figure 9 presents the execution time overheads involved by our approach based on the longest execution path of DeckX, traversing more than forty components from the application level as well as the container and RTOS levels. The testing environment consists of a Pentium 4 monoprocessor at 2.0 GHz, running 100,000 times the execution path[6]. These results show that the involved overheads are completly negligible for the `Flattened` scenario (Fig. 9(d)) and acceptable in the other cases (Fig. 9(b) & (c)).

	Ref.	Component-Based Design		
	(a)	(b) `Flexible`	(c) `Not flexible`	(d) `Flattened`
Mean (μs)	176.0	**+2.9 %**	**+1.7 %**	**+0.3 %**
Std. dev.	7.2	7.2	7.3	7.2

Fig. 9. Execution Time Overheads (and Standard Deviation).

7 Related Work

Specializing Component Models with Annotations. In programming languages, the use of annotations is widely applied to specialize their basic constructs. However, to the best of our knowledge, only the Think ADL [11], which we drew our inspiration from, and UML2 [20] exploit this feature to specialize architectural constructs. However, with Think, their uses are limited to optimization properties, configuring the last stages of the Think's code generation process, and not to refine the applicative architecture with other non-functional concerns. In turn, UML2 defines the *composite structure diagram* for specifying software architectures, and introduces the notion of profiles [8]. The latter is the built-in lightweight mechanism that serves to customize UML artifacts for a specific domain or purpose via *stereotypes*. Thus, the latter could be used to extend the semantics of the composite structure diagram artifacts. Our approach shares with UML the notion of annotation, close to the one of a stereotype.

[5] The total does not equal to 100% due to some code and data generated in the binary by the linking process of the C compiler.

[6] The scenario was "simulated" under a Linux 2.6 kernel (using a Linux port of μC) patched by Rt-Preempt. The latter converts the kernel into a fully preemptible one with high resolution clock support, allowing precise performance measures.

Extensible Container-Based Approaches. Even if component containers – originating for the EJB – are a key part of component frameworks, they generally support a predefined set of non-functional concerns. On the contrary, the PIN component model [17] is based on generative techniques to generate custom containers encompassing component interactions and implementing non-functional concerns. A strong separation of concerns is preserved between the latter and the code implemented by applicative components. Despite these similarities with our contribution, PIN relies on a code-centric approach (based on AspectC++) for generating containers, whereas Hulotte capitalizes on the component paradigm at this level. In this respect, Asbaco [16] and AoKell [23] are similar to our approach, both targeting the Java language for IT systems. However, they rely on costly mechanisms such as load-time mixin technique based on bytecode generation not suitable for embedded systems, and do not consider applications constrained by time and requiring low-level services from the operating system.

Handling Flexibility in Embedded Software Stacks Based on CBSE. Considering the approaches targeting real-time embedded systems, CBSE has been adopted either at operating system level [7, 10, 15, 22] or to propose Architecture Description Languages capturing the domain's relevant abstractions [1, 6, 9, 24]. In the first case, CBSE is exploited to provide a set of components used as building blocks to configure an operating system. However, within these approaches, the applicative components directly use the services provided by the OS, without any intermediate and flexible layer implementing non-functional concerns in an oblivious manner for the application developer. On the contrary, software stacks provided by the mentioned ADLs propose domain-specific abstractions implemented by a dedicated runtime, but do not provide an engineering of this layer to support new features not initially supported by their languages.

8 Conclusion

As embedded systems must be constantly adapted to support new features with a growing complexity, it is becoming necessary to use current software engineering principles and methodologies to increase software productivity. CBSE is widely-known to foster separations of concerns, reuse, and flexibility to shorten development time.

This paper presents a three-tier approach for composition of flexible real-time embedded software stacks. It relies on a design process where flexibility is achieved by two features: *i)* an annotation-based mechanism for specializing the application's architectural artefacts with non-functional concerns, and *ii)* a container model for the generation and composition of the runtime platform implementing them. The extensible nature of the containers makes them suitable for implementing complex features supported by the platform, relieving the application developer from dealing with redundant and error-prone tasks. Containers offer a straightforward design space for adapting the platform towards various application requirements. Moreover, our software stack model relies on a

component-based RTOS which provides, by composition, the low-level services strictly required by the upper layers. In this paper, we apply our approach for designing and implementing real-time audio applications. We demonstrate through a real-life case study that the impact on performances of our design process is negligible.

As a future work, we plan to provide a richer library of annotations encompassing various communication models and execution models (*e.g.*, *Constant Bandwidth* or *Contract-Based* Scheduling Servers) commonly used in RTE systems. Moreover, we envision to extend the generic component model on which the software stack relies with a behavioral model based on automata. The composition of these automata gives the global behavior of the whole stack outputed by our process, including low-level OS primitives, analyzable by model-checking tools (such as deadlock-free analysis).

Acknowledgment

This work was supported by the french Minalogic MIND project.

References

1. Automotive Open System Architecture (AUTOSAR), `http://www.autosar.org`.
2. *MicroC/OS-II: The Real-Time Kernel*. CMP Media, Inc., USA, 2002.
3. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL Component Model and its Support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Software Practice & Experience*, 36(11–12):1257–1284, 2006.
4. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd ed.* Addison-Wesley, 2002.
5. J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86, jun 2002.
6. P. H. Feiler, B. Lewis, S. Vestal, and E. Colbert. An overview of the SAE Architecture & Design Language (AADL) Standart : A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *Architecture Description Language, workshop at IFIP World Computer Congress*, 2004.
7. B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit A Substrate for Kernel and Language Research. *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, 1997.
8. L. Fuentes and A. Vallecillo. An Introduction to UML Profiles. In *UPGRADE, The European Journal for the Informatics Professional*, pages 5–13, April 2004.
9. H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren. SaveCCM - A Component Model for Safety-Critical Real-Time Systems. In *EUROMICRO'04: Proceedings of the 30th EUROMICRO Conference*, pages 627–635, Washington, DC, USA, 2004. IEEE Computer Society.
10. P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. *Ambient Intelligence*, pages 115–148, 2005.

11. O. Lobry, J. Navas, and J.-P. Babau. Optimizing Component-Based Embedded Software. *Int. Conf. on Computer Software and Applications*, 2:491–496, 2009.
12. O. Lobry and J. Polakovic. Controlling the Performance Overhead of Component-Based Systems. In *Software Composition*, pages 149–156. Springer, 2008.
13. F. Loiret, M. Malohlava, A. Plšek, P. Merle, and L. Seinturier. Constructing Domain-Specific Component Frameworks through Architecture Refinement. In *35th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA'09)*, pages 375–382, Aug. 2009.
14. F. Loiret, J. Navas, J.-P. Babau, and O. Lobry. Component-Based Real-Time Operating System for Embedded Applications. In *Proceedings of the* 12^{th} *International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'09)*, volume 5582 of *Lecture Notes in Computer Science*, pages 209–226, East Stroudsburg, Pennsylvania, USA, jun 2009. Springer.
15. A. Massa. *Embedded Software Development with eCos*. Prentice Hall, 2002.
16. V. Mencl and T. Bures. Microcomponent-Based Component Controllers: A Foundation for Component Aspects. *Asia-Pacific Software Engineering Conf.*, 0:729–737, 2005.
17. G. A. Moreno. Creating Custom Containers with Generative Techniques. In *5th Int. Conf. on Generative Programming and Component Engineering (GPCE'06)*, pages 29–38. ACM, 2006.
18. C. Noguera and F. Loiret. Checking Architectural and Implementation Constraints for Domain-Specific Component Frameworks using Models. In *35th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA'09)*, pages 125–132, Aug. 2009.
19. OMG. *UML 2.0 Object Constraint Language (OCL) Specification*.
20. OMG. Object Management Group: Unified Modeling Language – Superstructure Version 2.1.1, 2007.
21. A. Plšek, F. Loiret, P. Merle, and L. Seinturier. A Component Framework for Java-based Real-time Embedded Systems. In *Proceedings of the* 9^{th} *ACM/IFIP/USENIX International Conference on Middleware (Middleware'08)*, pages 124–143, Leuven, Belgium, dec 2008. Springer-Verlag.
22. A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component Composition for Systems Software. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, 2000.
23. L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A Component Model Engineered with Components and Aspects. In *9th Int. Symp. on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *LNCS*, pages 139–153. Springer, 2006.
24. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.